Requirement Verification with CRML and OpenModelica

Lena Buffoni[1] Adrian Pop[1] Audrey Jardin[2]

[1]Department of Computer and Information Science, Linköping University, Sweden,
{lena.buffoni,adrian.pop}@liu.se
[2]R&D, Electricité de France, France, audrey.jardin@edf.fr

Abstract

Common Requirement Modeling Language (CRML) is a language designed to express requirements in an intuitive manner, in this paper we present the recent advancements in tool support for the requirement modeling and verification workflow in OpenModelica and illustrate this on the Traffic Light use-case.
Keywords: CRML, Requirements, Verification, Complex Systems, Digital Engineering

1 Introduction

Systems Engineering is *a transdisciplinary and integrative approach to enable the successful realization, use and retirement of engineered systems* (INCOSE Fellows 2019).

Thanks to an approach centered on requirements, its main benefits lie in better consideration of the system's environment (in terms of coordination of the various stakeholders, support of associated constraints and interfaces with other systems). It also provides an iterative approach making it possible to guide choices towards solutions "just needed" throughout the life cycle of the system. Impact analyses of some changes during the system development cycle (e.g. coming from regulatory tightening, partner entry and/or withdrawal, etc.) can be performed early in the design process and knowledge can be better capitalized to refine the product strategy in its whole from one implementation project to another.

Despite numerous success stories, in particular, in the most advanced systems engineering industries such as defense and aerospace, many industries still struggle to implement systems engineering practically (Bretz, Kaiser, and Dumitrescu 2019; Kiniry et al. 2022).

In addition to budgetary and socio-organizational difficulties linked to change management, one of the major pitfalls often encountered is to limit the management of requirements to the documentary aspect. Technical contributors then perceive requirement management as an additional task to their usual engineering activities and with a relatively poor added value at the very limited time scale of their responsibility in the project compared to the lifespan of the system or even the company. In the worst case, this can generate a certain disillusionment and at least partial disengagement of some of the contributors.

When verification and validation of requirements are taken into account correctly in the organization and in the projects, the second flaw lies in the difficulty of finding "off-the-shelf" software solutions to handle requirements "from end to end" during the system lifecycle. In particular, beyond documenting the traceability of requirements and their theoretical correspondence to design elements (such as requirement management tools like IBM Doors, Polarion, etc.), using requirements as a guide for the development process requires the ability to capitalize this kind of industrial constraints in the form of models to make this knowledge usable by a computer (i.e. calculable) and to allow the automation of a certain number of tasks such as checking the consistency of a set of requirements, verifying the compatibility of a solution with a set of regulations, studying the impact of changes in the specifications or in the design, etc.

As defined by INCOSE in (SEBoK v2.11 2024), a requirement is a "*statement that identifies a system, product, or process characteristic or constraint, which is unambiguous, clear, unique, consistent, stand-alone (not grouped), and verifiable, and is deemed necessary for stakeholder acceptability*" and six verification methods are usually acknowledged:

1. *Inspection* requires a human sense or a simple measurement method to examine the object studied once it exists.
2. *Test* consists in performing some verifications on the existing object, often with dedicated instrumentation, and under specific controlled conditions.
3. *Demonstration* checks the desired characteristics of the existing object under its *classical* operating conditions.
4. *Sampling* tests also characteristics, but not only on one realization of the object studied but on a sample of several realizations.
5. *Analogy* uses the similarity of the object studied with elements for which the desired characteristics have already been proven.
6. *Analysis* shows theoretical compliance of the object studied using analytical evidence based on mathematical reasoning (calculation, modeling and / or simulation).

The article focuses here on the verification method based on analysis, which is the only option one can have for checking objects early in their design process or when testing or demonstration in real-life conditions is not possible (e.g. for cost reasons) or even not desirable (e.g. for safety reasons). Several approaches already exist in the literature to formalize requirements and their verification such as LTL (Linear Temporal Logic) and CTL (Computation Tree Logic) (Baier and Katoen 2008) , timed (Alur 1999) and hybrid (Henzinger 2000) automata or UML/SysML state behavioral diagrams (OMG 2017; OMG 2025b). (Bouskela, Buffoni, et al. 2023) explains why existing solutions are not really suitable for complex systems with strong physical aspects, the origin of the Common Requirement Modeling Language (CRML) from ITEA MODRIO project previous work's on the FORM-L language and how such a new language separate from Modelica could be a possible solution. In short, the existing formal requirements modeling methods have the following disadvantages: they lack of object-orientation, have difficult mathematical syntax, often do not deal with probabilistic aspects and, most importantly, tend to use abstractions of the system in the form of state machines, which already express a kind of a behavior and is not appropriate to correctly deal with systems with strong physical aspects that evolve continuously over time.

The novelty of the current paper is to show the recent progress made for supporting CRML in OpenModelica and how this new feature can practically help newcomers automating the verification of some temporal requirements with the use of Modelica/FMI (or more largely black box) models.

Section 2 briefly presents the CRML language and the associated methodology to verify requirements through the simulation of solution models. Section 3 provides details on how the continuous- and discrete-time domains are handled to express dynamic requirements in CRML. Section 4 introduces the Traffic Light System (TLS) used as a common thread example to illustrate the CRML approach. Section 5 describes the current tooling in OpenModelica. Section 6 concludes with some perspectives for improving the overall toolchain and its potential integration into a larger (Model-Based Systems Engineering) MBSE framework.

2 CRML in a Nutshell

CRML is a language for verifying temporal requirements on systems with strong physical concerns and for which the dynamical interactions with its environment and interfaced systems are of most importance.

The main goal of CRML is to be a high-level language enabling the definition of requirements in quite readable but computable expressions and serving as a pivotal format capable of targeting different verification engines either by simulation (as documented here with its integration in OpenModelica) or by formal proof (future work not documented here about connection with model checking approaches).

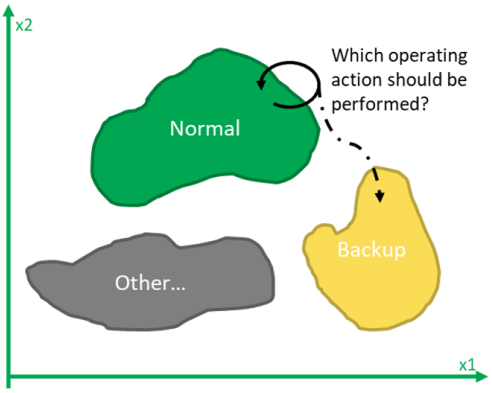


Figure 1. A typical realistic dynamic requirement.

The following paragraphs briefly recap how a requirement could be expressed and verified using CRML. Appendix gives a quick overview of the CRML built-in elements and most useful operators. Interested readers can refer to the online tutorial (Buffoni et al. 2023) and the language specifications (CRML v1.2 2023) for more details.

2.1 Requirement Expression

A requirement in CRML is defined as an expression combining up to 4 items:

- a **condition** to be checked;
- a **spatial locator** indicating on which object the condition has to be verified;
- a **time locator** defining when the condition has to be satisfied;
- (optionally) a **statistical target** to indicate with which performance the condition has to be satisfied.

A CRML requirement is therefore of a "special" Boolean built-in type, here called Boolean4 stating that a requirement value at instant `t` can vary among the set `{true, false, undefined, undecided}`. This is illustrated with the Traffic Light System example of Section 4. The theoretical rationale for introducing these 4 values could be found in (Bouskela and Jardin 2018) where a new temporal logic algebra is defined to rigorously compute the value of requirement over continuous-time periods and to enable the combination of requirements together.

One may consider that checking requirements for physical systems always amounts more or less to checking whether main variables of interest do not exceed thresholds. The main difficulties that make the handling of requirements more complex are as follows:

DOI 10.3384/ecp218971

Table 1. CRML pattern for a typical realistic dynamic requirement.

Req.	*Natural Language*	*CRML Formalization*
r1	During operation, the system should stay within its normal domain.	`Requirement r1 is 'during' inOperation 'ensure' inNormalDomain;`

r2	If the system fails to stay within its operating domain, then it should not stay outside of its normal domain for more than x minutes.	`Requirement r2 is 'during' inOperation 'ensure' (not inNormalDomain 'implies' r2_outside);`
		`Requirement r2_outside is 'during' [inNormalDomain 'becomes false', inNormalDomain 'becomes false' + x mn] 'check at end' b;`
		`Boolean b is inOperation 'implies' inNormalDomain;`

r3	The system should not go outside its normal domain more than n times per year.	`Requirement r3 is 'count' ((b 'becomes false') on [b 'becomes false', b 'becomes false' + 1 year]) <= n;`
r4	If (r1 and r2 and r3) fail, then the system should go to its backup domain within y minutes as soon as the failure is detected.	`Requirement r4 is not (r1 and r2 and r3) 'implies' 'during' [(r1 and r2 and r3) 'becomes false', (r1 and r2 and r3) 'becomes false' + y mn] 'check at end' inBackupDomain;`
R	(**The "real" complete requirement:**) During system operating life, r1 and r2 and r3 and r4 should be satisfied with a probability of success of p.	`Real prob is estimator Probability (r1 and r2 and r3 and r4) at inSystemOperatingLife 'becomes false';` `Requirement R is 'during' inSystemOperatingLife 'check at end prob' > p;`

- The authorized values vary dynamically depending on the system operating modes and the periods of time.
- Time response and/or probabilistic criteria should be added to make the requirements achievable by real systems.
- The number of requirements is rapidly growing (due to the number of stakeholders, the complexity of the system/subsystems/components, and the possible evolution over the system lifespan).

The CRML language enables alleviating these problems by allowing the formal expression (Table 1) of realistic dynamical requirements (Figure 1).

Listing 1. Example of user-defined operator in CRML

```
Operator [ Periods ] 'during' Boolean b =
    Periods [ new Clock b, new Clock not b
    ];
```

The user can enhance the readability of CRML statements by defining his own operators with a chosen syntax simply by combining built-in operators. For example, the `during` operator in Table 1 has been defined as a time period constructor from a Boolean input (Listing 1). This `during` operator is part of a library known as FORM-L used in CRML to define time periods and conditions constructors that are frequently encountered (see Appendix for more examples of custom operators).

2.2 Requirement Verification

As stated above, a CRML requirement is a Boolean4 variable. Its value is: `true` (resp. `false`) if the condition is satisfied (resp. violated) for the defined time locator, `undefined` if the time locator has not been tested over the test scenario, `undecided` if the test scenario has finished before a decision could be made (i.e., before the condition has been violated or before the end of the time locator).

The value of a CRML requirement is computed from verification models made of four different kinds of model (Figure 2):

- the *behavioral model* capturing the behavior of design solutions;
- the *requirement model* defining envelopes of acceptable behaviors;
- the *binding model* mapping the variables of the two previous models and processing the required conversions (the functional variables mentioned in the requirements could have different naming convention and units than the physical quantities computed in the behavioral model);
- the *test model* specifying the operating scenarios to be studied.

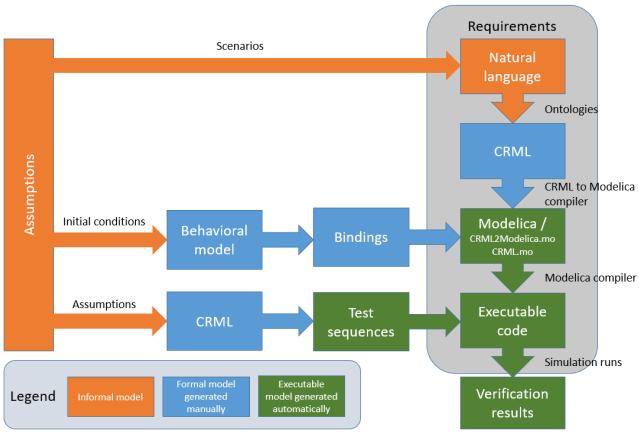


Figure 2. Requirement verification workflow.

3 Time Management in CRML

The main temporal constructs in CRML are Clock, Event, Period and Periods.

The CRML2Modelica library contains the definitions of basic types in CRML, including events and periods. These definitions are split into two parts, a record structure that can then be passed along an argument in subsequent equations, and a model containing the equations for the record.

In Modelica Events are represented by a trigger (a 4-valued boolean) and a time of occurrence.

A single period, as shown in Listing 2 is defined by opening and closing events, while a set of periods is defined by clocks or sets of opening and closing events (leading to possibly overlapping time periods).

For example, a new Period instance is generated and instantiated as follows:

Listing 2. Period instance creation in Modelica

```
record CRMLPeriod
 Boolean isLeftBoundaryIncluded "If true,
     the left boundaries of the time
     periods are included";
 Boolean isRightBoundaryIncluded "If true,
     the right boundaries of the time
     periods are included";
 public
  Types.Event start_event;
  Types.Event close_event;
  Boolean is_open;
end CRMLPeriod;

model CRMLPeriod_build
 CRMLPeriod P;
 equation
 P.is_open =
  if( (CRMLtoModelica.Functions.
      Event2Boolean (P.start_event) ==
      Boolean4.true4) and not (
      CRMLtoModelica.Functions.
      Event2Boolean (P.close_event)==
      Boolean4.true4))
  then true else false;
```


DOI
10.3384/ecp218971

```
end CRMLPeriod_build;
```

4 Traffic Light Use-Case

The purpose of the Traffic Light example is to illustrate how CRML could be used in the OpenModelica framework from the requirement formalization phase to verification by simulation and analysis phases.

This example was chosen to show, on a system known to all, how some realistic requirements could be handled in a very practical way with CRML and OpenModelica. It is deliberately simplified to focus on the software tooling part.

The mission of the Traffic Light System (TLS) is to avoid accidents and fluidize traffic by minimizing congestion. The main functions are, respectively: F1) give clear instructions to each vehicle and F2) prioritize crossing durations based on road traffic. The corresponding requirements could therefore check that *"at most only one light should be on"* and *"the time spent in a light mode must be between two limits"*. A possible formalization in CRML is given in Listing 3.

Listing 3. Requirements on the Traffic Light System

```
model TrafficLightSpecification is
 // Import of libraries
 flatten {ETL, FORM_L}
 union {
 // List of external variables
 Boolean red is external;
 Boolean yellow is external;
 Boolean green is external;

 Boolean operation is external;
 Boolean night_mode is external;
 Boolean day_mode is not night_mode;

 // Definition of requirements
 // req0: "During operation, no more than
    one light should be on (a flashing
    mode could be specified later)"
 Boolean{} lights is {red, yellow, green};
 Integer n_lights_on is card { filter
    lights (element == true) };

 Requirement req0 is
  ('during' operation) ('check anytime' (
     n_lights_on <= 1) );

 // Day_mode
 // req1: "After green, next step is yellow
    "
 Requirement req1 is
  (( 'after' ( green 'becomes true' ) '
     before' ( yellow 'becomes true' ))
     while day_mode)
  ('check count' (red 'becomes true') '=='
     0);

 // req2: "Step green should stay active
    for at least 30 seconds"
 Requirement req2 is
  (( 'after' (green 'becomes true') 'for'
     30) while day_mode)
  ('ensure' green);

 // req3: "After green becomes active + 30
    seconds, next step should turn yellow
    within 0.2 seconds"
 Requirement req3 is
  (( 'after' (green 'becomes true' + 30) '
     for' 0.2) while day_mode)
  ('check at end' yellow);

 // Night_mode
 // req4: "During night, yellow should only
    be used"
 Requirement req4 is
  ('during' night_mode)
  ('check count' ( (red or green) 'becomes
     true') '==' 0);
 // req5: "During night, yellow should
    flash every 2 seconds"
 Requirement req5a is
  ('during' night_mode)
  ('check count' ( yellow 'becomes true') '
     <>' 0);
 Requirement req5b is
  ('after' ( yellow 'becomes true') 'within
     ' 2) while night_mode)
  ('ensure' yellow);
 Requirement req5c is
  ('after' ( yellow 'becomes true') 'for'
     2) while night_mode)
  ('check at end' not yellow);
 Requirement req5 is req5a and req5b and
    req5c;
};
```

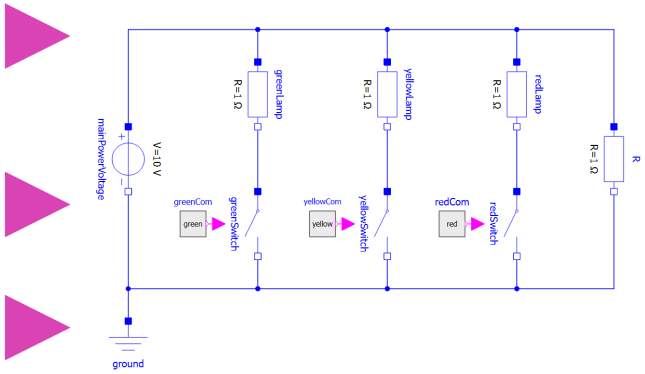


Figure 3. Traffic Light implementation as a circuit.

The physical behavior of TLS could be modeled as an electronic circuit that activates three different lights (Figure 3). The control part of TLS could be modeled in two different ways: either as a grafcet or as a finite-state machine using the synchronous built-in operators of the Modelica language as shown in Figure 4.

4.1 CRML Compilation

The CRML2Modelica compiler is a proof of concept compiler written in Java. It is open source and available on

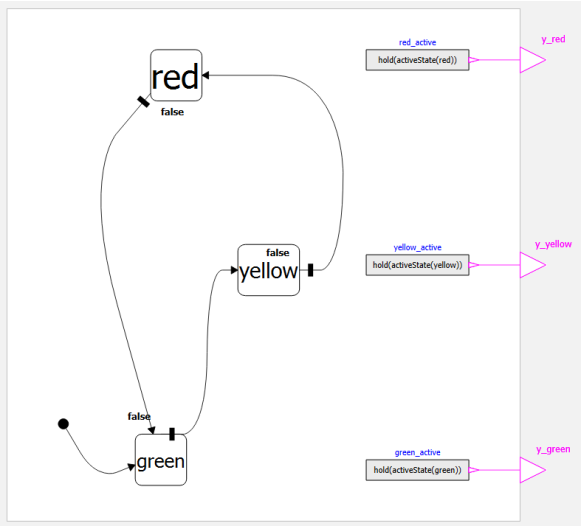


Figure 4. Traffic Light control implemented as a state machine.

GitHub [1]. Currently it supports almost all the Extended Temporal Library (ETL) library blocks which define all the operators required for computing Boolean4-status of a requirement.

The CRML2Modelica compiler goes through the CRML file once first, to retrieve all signatures of user-defined operators and category's definitions and then a second time translating all the definitions. In addition to temporal aspects, three CRML features require special handling: Boolean4, categories and sets.

Boolean4

Boolean4 is defined in Modelica as an enumeration as shown in Listing 4.

Listing 4. Boolean4 translation in Modelica

```
type Boolean4 = enumeration(undefined,
    undecided, false4, true4);
```

Categories

Categories are applied directly to the operator definitions, and the corresponding operators are replaced. For example, the operator `set to false` in Listing 5 is translated to Modelica as shown in Listing 6 and the operator `id` is replaced by `cte_false` as specified by the category `varying1`.

Listing 5. Category in CRML

```
Operator [ Boolean ]  'id'  Boolean b = b;
Operator [ Boolean ]  'cte_false'  Boolean
    b = false;
Category varying1 = { ('id', 'cte_false')
    };
Operator [ Boolean ] 'set to false' Boolean
     b = apply varying1 on ( 'id' b );
        ...
Boolean b_varying1_on_id_true is 'set to
    false' b_true; // value should be false
```

[1] https://github.com/OpenModelica/CRML

Listing 6. Category translation to Modelica

```
model 'set to false'
 output CRMLtoModelica.Types.Boolean4 out;
 input CRMLtoModelica.Types.Boolean4 b;
 'cte_false' 'cte_false0'(b=b);
 equation
 out ='cte_false0'.out;
end 'set to false';
```

Sets

Sets are handled as fixed size arrays in the current implementation, for sets with a known number of elements, the array size is set accordingly, for sets with an unknown size, the array size is set to a predefined buffer size. Periods are a special case of this and are currently mapped to a record with an array with a static size, but if a larger number of events is expected, the buffer size can be increased with a flag to the compiler.

In our Julia-based OpenModelica framework (Tinnerholm, Pop, and Sjölund 2022) we support automatic recompilation during simulation and one can increase/shrink the size of an array or change any other aspect of the model. An example of a `when` equation triggering such a change is given in Listing 7. Using this functionality in the future, we can make the translated Modelica code more flexible and more efficient as resizing can be tailored to the needs of each variable.

Listing 7. Dynamic Array Resize

```
when requestedSize > N then
  // Recompilation on parameter change
  recompilation(N /* what to change */,
      requestedSize + 100 /* value */);
end when;
```

4.2 Requirement Verification

A requirement model is verified for a behavior model and a scenario. In this paper we focus on Modelica, but as a requirement model can be exported as an FMU, it could be verified against any other FMU representing a behavioral model.

In order to identify the inputs from the behavioural model, CRML uses `external` variables, that are variables that do not need to be instantiated within the model. A list of external variables can be exported during compilation into a file. This list can then be fed into a semi-automated bindings algorithm for mapping requirements to the behavioural model.

A verification scenario that instantiates the requirement and behavioural models and connects them together can then be simulated and the results analysed.

The code generated for requirements is composed together with the unmodified Modelica model and the external variables in the CRML are bound to the Modelica variables. Usually this results in a balanced Modelica model but is certain cases, usually with badly specified CRML it might be possible to build invalid models.

DOI
10.3384/ecp218971

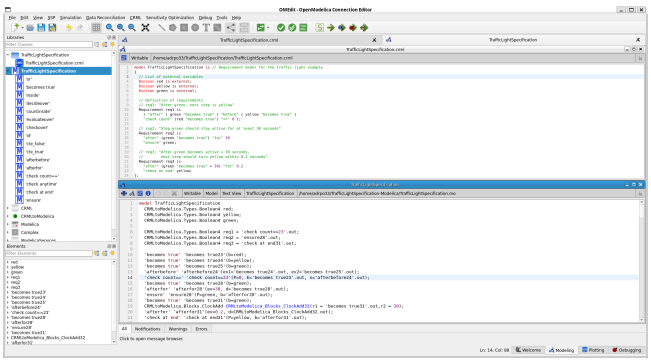


Figure 5. CRML file display in OMEdit and its Modelica translation.

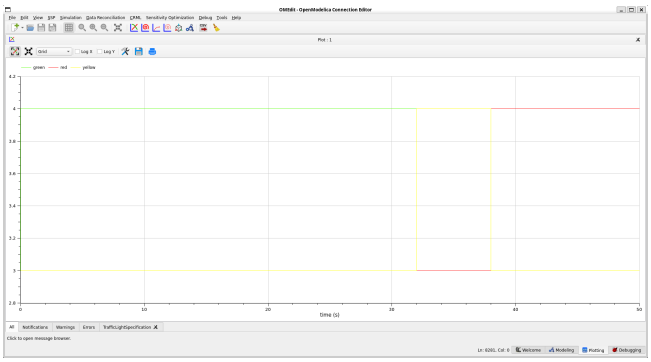


Figure 6. Simulation of the generated Modelica file corresponding to the original CRML file.

5 CRML Support in M&S tools

We have started integrating the CRML tools into OpenModelica. The CRML files can be loaded in the OpenModelica Connection Editor (OMEdit) and the text displayed with syntax highlighting, see Figure 5, top part. It is also possible to load entire directories containing CRML files.

The CRML files can be translated to Modelica from OMEdit via menu actions (via right click) and the translated .mo files are automatically loaded into OMEdit, see Figure 5, bottom part.

As expected, the generated Modelica files can also be simulated, as in Figure 6.

To tell OMEdit where it can find the tools and required Modelica libraries for the CRML translation a settings dialog is used, see Figure 7. If additional Modelica libraries are needed, one can specify them here.

The CRML compiler also specifies a test suite with about 150 test cases based on the ETL and FORML library blocks as well as more complex use-cases. This test suite can be run command line or from OMEdit and generates a test report that allows to evaluate coverage progress. Currently around 30% of the tests are successful as the test suite also includes features that are not yet supported in the compiler.

Using CRML files as text is only the first integration step in OpenModelica. Work is ongoing to support a

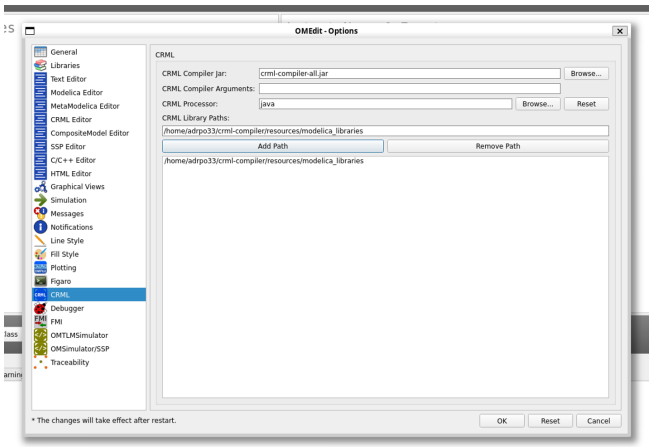


Figure 7. CRML tools settings in OMEdit.

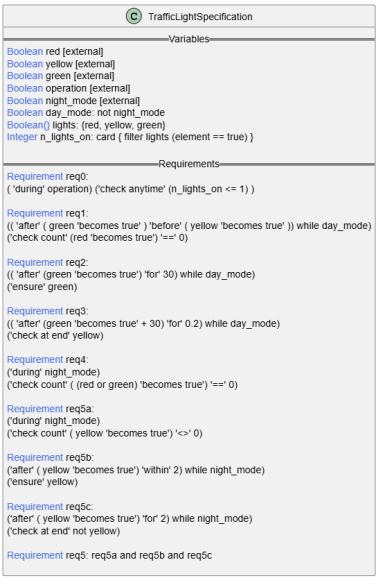


Figure 8. A graphical representation for CRML for the given example.

graphical representation language for CRML similar to UML as proposed by (Mazurié 2023), see Figure 8.

6 Conclusion and Future Work

CRML tooling is progressing to enable the formalization of realistic dynamic requirements of complex systems. Integration into OpenModelica enables a first step towards an easy access for the Modelica/FMI community to check requirements with simulation means. In order to be applicable to industrial-level use cases, future work plans to improve the toolset in several areas:

- Improvement of the user interface to visualise requirements simulation results as dashboards and structured html pages.
- Integration with the ReqIF standard (OMG 2025a) to provide a native gateway to the requirements management and traceability tools. This is in particular

needed to handle the metadata associated with the lifecycle of each requirement, coordinate the teams issuing requirements and deduce which set of requirements is valid and must be verified at a given point in time during the project.

- Alignment with the emerging SysML v2 standard (OMG 2025b) to structure and better visualise a large volume of requirements.

Other work is also planned, notably on the use of structural analysis techniques to statically verify consistency between CRML requirements and the introduction of the concept of Assume/Guarantee contract to facilitate the generation of test scenarios from CRML constraints.

7 Acknowledgements

The CRML initiative has initially been supported by the ITEA3 EMBrACE project, as well as National Research, Development and Innovation Fund of Hungary, financed under the [2019-2.1.1-EUREKA-2019-00001] funding scheme. It is continued and improved within the Open Source Modelica Consortium partly through EDF funding.

References

Alur, Rajeev (1999). "Timed automata". In: *International Conference on Computer Aided Verification*. Springer, pp. 8–22.

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of model checking*. MIT press.

Bouskela, Daniel, Lena Buffoni, et al. (2023). "The Common Requirement Modeling Language". In: *Proceedings of the Modelica Conference 2023*.

Bouskela, Daniel and Audrey Jardin (2018). "ETL: A New temporal language for the verification of cyber-physical systems". In: *Proceedings of the Annual IEEE International Systems Conference 2018*. DOI: 10.1109/SYSCON.2018.8369502.

Bretz, Lukas, Lydia Kaiser, and Roman Dumitrescu (2019). "An analysis of barriers for the introduction of Systems Engineering". In: *Proceedings of the 29th CIRP Design*. Elsevier, pp. 783–789.

Buffoni, Lena et al. (2023). *Tutorial: CRML A Language for Verifying Realistic Dynamic Requirements*. Tech. rep. MODPROD. URL: https://github.com/OpenModelica/CRML/tree/main/resources/crml_tutorial.

CRML v1.2 (2023). *Specification v1.2 of the Common Requirement Modeling Language*. Tech. rep. ITEA EMBrACE Project. URL: https://github.com/OpenModelica/CRML/blob/main/language_specification/CRML%20specification_v1.2.pdf.

Henzinger, Thomas A (2000). "The theory of hybrid automata". In: *Verification of digital and hybrid systems*. Springer, pp. 265–292.

INCOSE Fellows (2019). *Briefing to INCOSE Board of Directors*. Tech. rep. INCOSE.

Kiniry, Joseph et al. (2022). *High Assurance Rigorous Digital Engineering for Nuclear Safety (HARDENS) Final Report*. Tech. rep. Galois.

Mazurié, Baptiste (2023). "Start of a new design method for a competitive Small Modular Reactor (SMR) adaptable to future uses". MA thesis. KTH, School of Engineering Sciences (SCI).

OMG (2017). *Unified Modeling Language (UML) version 2.5.1*. URL: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 2025-07-29).

OMG (2025a). *Requirements Interchange Format (ReqIF)*. URL: https://www.omg.org/reqif/ (visited on 2025-07-31).

OMG (2025b). *Systems Modeling Language (SysML) version 2*. URL: https://github.com/Systems-Modeling/SysML-v2-Release/blob/master/doc/2a-OMG_Systems_Modeling_Language.pdf (visited on 2025-07-29).

SEBoK v2.11 (2024). *Guide to the Systems Engineering Body of Knowledge*. https://sebokwiki.org/wiki/.

Tinnerholm, John, Adrian Pop, and Martin Sjölund (2022). "A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability". In: *Electronics* 11.11, 1772. DOI: 10.3390/electronics11111772.

Appendix: CRML Cheat Sheet

Key Concepts

Requirements

A requirement is a `Boolean4` combining up to 4 items:

```
R = (Where) (When) (What) [How well]
```

Multiple requirements can be combined according to the algebra defined on the `Boolean4` type.

b	true	false	undecided	undefined
not b	false	true	undecided	undefined

b1 and b2	**true**	**false**	**undecided**	**undefined**
true	true	false	undecided	true
false	false	false	false	false
undecided	undecided	false	undecided	undecided
undefined	true	false	undecided	undefined

Syntax

Notation

```
[ expr ] ..... optional expression
{ expr } ..... expression repeated
expr_1 | ... | expr_n .... n expressions
'char' ..... character
"string" ..... string
```

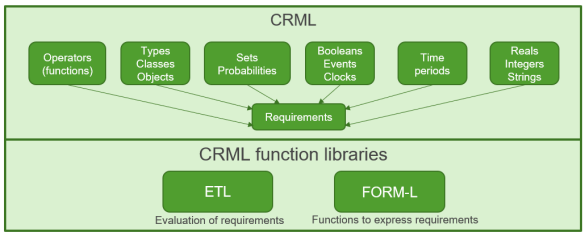
Expressions

```
[ [type] ident is] [ value |
external ] [ ; | , ] ..... expression
// This is single-line ..... comment
/* This is a multi-line */ .. comment
```

Keywords

Types: Boolean, Category, class, Clock, Event, Integer, library, model, Operator, package, Period, Periods, Probability, Real, String, Template, type
Special values: false, true, undecided, undefined, time
Special characters: (,),[,], {, }, ,, ;, ., ", ', E, e, //, /*, */, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Operators: =, +, -, *, /, <, <=, >, >=, ==, <>, ^, acos, alias, and, asin, associate, at, card, constant, cos, duration, element, else, end, estimator, exp, extends, external, filter, flatten, forbid, if, integrate, is, log, log10, mod, new, not, on, or, parameter, partial, proj, redeclare, sin, start, then, tick, time from, union, variance, while, with

Architecture



Real Operators

`Real x is decimal_value ....` constructor
`Real x is new Real n ...` constructor from *Integer n*
`x1 + | - x2 .....` binary addition | subtraction
`+ | -x1 ..........` unary addition | subtraction
`x1 * x2 .....` multiplication
`x1 / x2 .....` division
`x1^x2 .....` exponentiation

`if b then x1 else x2 ..........` if-clause
`x1 at c .....` value at *Clock c*
`duration b on P .....` duration during which *Boolean b* is true over a *Period P*
`e2 - e1 ...` elapsed duration between two *Event*
`time frome e1 ...` elapsed time from an *Event*

Integer Operators

`Integer n is integer_value` constructor
`Integer n is new Integer x` constructor from *Real x*
`n1 + | - n2 .....` binary addition | subtraction
`+ | - n1 .....` unary addition | subtraction
`n1 * n2 .....` multiplication
`n1 / n2 .....` division
`n1^n2 .....` exponentiation

`if b then n1 else n2 ..........` if-clause
`n1 at c .....` value of *Integer n1* at *Clock c*
`card c .....` number of ticks of *Clock c*
`card S .....` number of elements of *set S*

String Operators

`String s is string_value ...` constructor
`String s is new String x ...` constructor from *Real* | *Integer* | *Boolean* x
`s1 + s2 .....` concatenation

Boolean Operators

`Boolean b is true | false | undecided | undefined .....` constructor
`Boolean b is new Boolean c` constructor from *Clock c*
`b1 and b2 .....` conjunction
`b1 or b2 .....` disjunction
`not b .....` negation
`b1 * b2 .....` filter
`b1 + b2 .....` accumulation
`integrate b1 on P` integration over a *Period*
`b1 == b2 .....` equality
`if b then b1 else b2 ..........` if-clause
`b1 at c .....` value at *Clock c*
`x1 > x2` strictly greater than for *Real* | *Integer xi*
`x1 < x2 ...` strictly less than for *Real* | *Integer xi*
`n1 >= | <= n2 .....` comparison of *Integer ni*
`n1 <> n2 .....` different from for *Integer ni*
`x1 == x2 .....` equality for *Integer* | *Type xi*
`e1 < | <= e2 ....` (strictly) before for *Event ei*
`e1 > | >= e2 .....` (strictly) after for *Event ei*

Event Operators

`Event e is new Event b .` constructor from first occurence of *Boolean b*
`e1 proj c .....` projection on ticks of *Clock c*
`e1 proj(d) c ..` projection for duration *Real d*
`e1 + d .....` delay of *Real d*
`tick c .....` current tick of *Clock c*
`p start .....` opening event of *Period p*
`p end .....` closing event of *Period p*

Clock Operators

`Clock c is new Clock b .` constructor from *Boolean b*
`c1 proj c2 ....` projection on ticks of *Clock c2*
`c1 proj(d) c2 .` projection for duration *Real d*
`c1 + d .....` delay of *Real d*
`c1 filter cond tick .....` filter

`c1 and c2` ..... conjunction
`c1 or c2` ..... disjunction

Period and Periods Operators

`Period P is [|] e1, e2 [|]` constructor from *Event*
`Periods P is {P1, P2, ..., Pn}` . constructor from a set of *Period*
`Periods P is [ | ] c1, c2 [ | ]` constructor from *Clock*
`P2 while P1` ..... filter

Probability Operators

`Probability px is new Probability b` ..... constructor from *Boolean b*
`Probability px is new Probability b at c` ..... constructor at ticks of *Clock c*
`estimator px` ..... estimator
`estimator variance px` . variance estimator

Sets Operators

`[T{} S is]{u1,...,un}` .. non-empty typed set
`[T{} S is]{}` ..... empty typed set of type T
`T{} S is {expr}` special set with elements of different types (=*class, model, library, or package*)
`S1 union S2` ..... union of sets
`flatten S1` ..... flattening of a set
`filter(S1 cond (element))` ..... filter

Customisation

Type Constructors

`type T2 is (T1 x is value(a1,..., an)){Type D1, ...Dn; D1 a1 [is value];...}` ..... custom type
`partial type T is value` .... partial type
`T1 alias s` .... renaming type T1 with *String s*
`T1 forbid f1,...,fn` forbidding operators *fi*

Operator Constructors

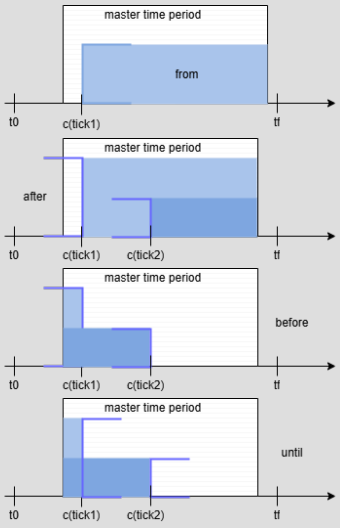
`Operator [T] 'w1' T1 u1 ... 'wn' Tn un = expr` .. custom operator 'w1w2wn' on inputs ui of type Ti
`Template w1 u1 ... wn un = expr` .... custom operator 'w1w2wn' on *Boolean*

Libraries

ETL Operators for Evaluating Booleans

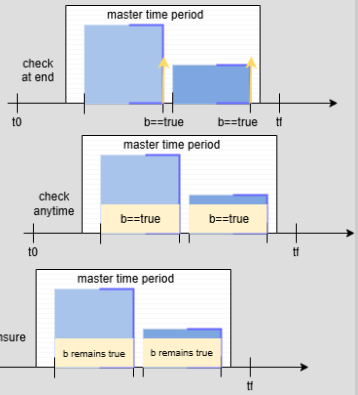
`c 'inside' p` .....
`b 'becomes true'` .....
`b 'becomes false'` .....
`b 'becomes true inside' p` .....
`b 'becomes false inside' p` .....
`'count' c 'inside' p` .....
`'decide' b 'over' p` . decision event, could be violation of *Boolean b* or end of *Period p*
`'evaluate' b 'over' p` accumulated value over a *Period p*
`'check' b 'over' P` accumulated value over a set of *Periods*

FORML Operators for Expressing Periods



`'from' c` .....
`'after' c` ....
`'before' c` ..
`'until' c` ....
`'during' b` .....
`'when' c` .....

FORML Operators for Expressing Conditions



`'check at end'`
`'check anytime'`
`P 'ensure' b`
`P 'check count' c '<=' n` .....

DOI 10.3384/ecp218971